(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 March 2001 (29.03.2001)

PCT

(10) International Publication Number
WO 01/22213 A2

(51) International Patent Classification[7]: G06F 9/00

(21) International Application Number: PCT/EP00/08976

(22) International Filing Date: 13 September 2000 (13.09.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
99402309.1 21 September 1999 (21.09.1999) EP

(71) Applicant: KONINKLIJKE PHILIPS ELECTRONICS N.V. [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(72) Inventor: RICCARDI, Fabio; Prof. Holstlaan 6, NL-5581 AW Eindhoven (NL).

(74) Agent: CHARPAIL, François; Internationaal Octrooibureau B.V., Prof. Holstlaan 6, NL-5581 AW Eindhoven (NL).

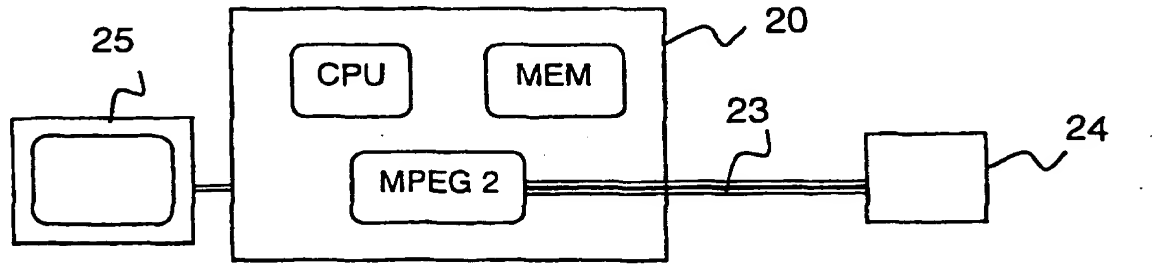
(81) Designated States *(national)*: CN, IN, JP, KR.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: OPTIMIZED BYTECODE INTERPRETER OF VIRTUAL MACHINE INSTRUCTIONS



(57) Abstract: The invention relates to a method of optimizing interpreted programs, in a virtual machine interpreter of a bytecode-based language, comprising means for dynamically reconfiguring said virtual machine with macro operation codes by replacing an original sequence of simple operation codes with a new sequence of said macro operation codes. The virtual machine interpreter is coded as an indirect threading interpreter thanks to a translation table containing the implementation addresses of the operation codes for translating the bytecodes into the operation code's implementation addresses. Application: embedded systems using any bytecode-based programming language, set to box for interactive video transmissions.

WO 01/22213 A2

Optimized bytecode interpreter of virtual machine instructions

FIELD OF THE INVENTION

The invention relates to run-time optimization of interpreted programs. It relates, more particularly, to a method for optimizing interpreted programs by means of a virtual machine which dynamically reconfigures itself with new macro operation codes. The invention applies to any bytecode-based programming language.

BACKGROUND OF THE INVENTION

Bytecode-based languages with programmer-visible stacks are popular as intermediate languages for compilers, and also as machine-independent executable program representations. They offer significant advantages for network computing. The article "Optimizing direct threaded code by selective in-lining", by I. Piumanta and F. Riccardi, in Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17, 1998, pp.291-300, describes a technique as mentioned in the opening paragraph, for optimizing interpreted programs. A virtual machine (VM) is used to interpret the programs thanks to a VM interpreter. The VM is a software implementation representing an architecture of a virtual processor on which applications especially compiled for this architecture are executed. The instructions of the virtual processor / machine are called bytecodes. The VM interpreter is the part of the VM which represents the bytecodes' execution mechanism. The bytecodes are said to be interpreted by the VM interpreter. The bytecodes' execution mechanism is currently implemented as an infinite loop with a switch case bloc. The technique described in the cited article applies to direct threaded interpreters. Threaded code interpreters execute the bytecodes in line. Each bytecode translation contains the reference to the next bytecode. Therefore, the bytecode translation as executed by a threaded interpreter does not involve the infinite loop. Even though threaded interpreters offer a performance advantage, they are too slow and require too much memory to be convenient for most embedded systems. In a direct threaded code interpreter, as described in the cited article, the VM bytecodes are represented with the address of their implementation, so that each bytecode can directly jump to the implementation of the next bytecode. A table is initialized before the translation operation.

with the addresses of each bytecode of the application in order that, when the bytecode translation takes place, the physical addresses of the bytecode implementations can be quickly accessible. The table allows to switch from a bytecode to another one. Direct threaded interpreters are rather fast but they involve code expansion. By changing bytecodes into direct threaded codes, the code size is increased by approximately 150%, because the operation codes are replaces with the addresses of their implementation code. In general, addresses need 4 bytes whereas the operation codes need only 1 byte. Therefore, direct threaded interpreters increase memory consumption and are thus not very suitable for embedded systems.

SUMMARY OF THE INVENTION

It is an object of the invention to provide a method for optimizing run-time of interpreted programs which is very convenient for embedded systems. Such systems may be, for example, satellite or cable transmission systems embedded into a digital video receiver, often called a set top box. But the invention also applies to any product whose operating system is based on a bytecode-based programming language. The invention also allows to save memory and CPU resources and can improve the performance of the system.

In accordance with the invention, it is described a method of optimizing interpreted programs in a virtual machine interpreter of a bytecode-based language, wherein the virtual machine dynamically reconfigures itself with new macro bytecodes (or opcodes) replacing sequences of simple bytecodes, and wherein the virtual machine interpreter is coded as a threaded code interpreter for translating the bytecodes into their implementation codes. The threaded code interpreter according to the invention is coded as an indirect threaded code interpreter thanks to a reference table which contains the implementation addresses of the bytecodes in order that during translation of a bytecode, the address of the next bytecode is retrieved to be able to jump to the next bytecode.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention and additional features, which may be optionally used to implement the invention, are apparent from and will be elucidated with reference to the drawings described hereinafter.

Fig. 1 is a bloc diagram illustrating the features of a method according to the invention.

Fig. 2 is a bloc diagram illustrating the features of a method according to the preferred embodiment of the invention.

Fig. 3 is a schematic diagram illustrating an example of a receiver according to the invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention will be now explained in greater detail, taking the Java language as an example, to illustrate a novel run-time optimization strategy applicable to any bytecode-based language.

The approach normally taken by Just-In-Time (JIT) compilers is to discard altogether the Java virtual machine (VM) interpreter and to translate the application's bytecode into native machine code prior to its execution (hence the Just-In-Time denomination). This process involves understanding the original application's semantic and re-expressing it into a more convenient native form. While this may be an efficient way of attaining performance, it also comes at the expense of a very large memory consumption in the one hand, because a bytecode-based language is more compact than a native code, and of large CPU (Central Processing Unit) resources in the other hand, because re-mapping Java bytecodes on the target machine is not an easy task.

The invention is also based on some sort of dynamic code generation, but its goal is not that of translating the application's Java bytecode into native machine code, but rather to dynamically adapt the Java VM to the execution of the application's specific bytecode sequences. The original application's Java bytecode is thus preserved, while the VM is dynamically enriched with novel bytecodes or operation codes (opcodes) improving its execution efficiency.

There are several advantages to this approach :

It does not increase the size of the executable code : the application is left into memory-efficient Java's bytecoded representation,

The VM's execution mechanisms is economic : there is only one execution mechanism, therefore the VM executing the application will not have to deal with multiple code representations which contributes to reduce its size and improve its reliability,

The code generation technique is rather simple: the VM optimizer has a very simple structure, the application's bytecode analysis is a one-pass table-driven procedure taking very little CPU resources, and which directly drives the synthesis of new bytecodes.

These properties make the invention suitable for embedded applications. The foundation of the optimization technique according to the invention lies in the study of the costs of the very basic mechanisms of an interpreter with respect to a category of "typical" applications. The relevance of the application's profile lies in the potential benefit attainable from the various optimization techniques that might be envisaged. Since the target is embedded applications, what might be define as "typical" applications are, for example, control applications, graphical user interfaces, and so forth.

It is assumed that the target applications are well mapped on the primitives offered by the underlying VM (object manipulations). Therefore, they will not benefit much from radical code transformations, but rather from a general improvement of the VM's execution mechanisms. To understand how to improve the efficiency of the VM, it was made use of Amdhal's law. In the version stated by Hennessy and Patterson, Amdhal's law is expressed as follows : "the performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used", or more synthetically : "make the common case fast".

Interpreter's performance depend on the representation chosen for executable code and on the mechanism used to dispatch the bytecodes. The first approach to reduce the implementation cost was to reduce the cost of instruction dispatching because the heart of an interpreter is its instruction dispatching mechanism. The typical interpreter, called pure bytecode interpreter, is implemented like a processor simulation : a large switch statement sitting in an endless loop, dispatching instructions to their implementations. Therefore, the inner loop of a pure bytecode interpreter is very simple : fetch the next bytecode and dispatch to the implementation using a switch statement. The interpreter is an infinite loop containing a switch statement to dispatch successive bytecodes, and passes control to the next bytecode by breaking out of the switch to pass control back to the start of the infinite loop. The following set of instructions illustrates an implementation of a typical bytecode interpreter.

```
Loop        (
            Op = *pc++;
            Switch (op) {
            Case op_1 :
                    // op_1's implementation
                    break;
            case op_2 :
```

```
          // op_2's implementation
          break;
    case op_3 :
          // op_3's implementation
          break;
    ...
    }
```

Assuming the compiler optimizes the jump chains from the breaks through the implicit jump at the end of the loop back to its beginning, the overheads associated with this approach are as follows :

increment the instruction pointer pc,
fetch the next bytecode from memory,
a redundant range check on the argument to switch,
fetch the address of the destination case label from a table,
jump to that address,
and at the end of each bytecode :
jump back to the start of the loop to fetch the next bytecode.

In this case the cost of instruction dispatching, ignoring all other sources of inefficiency such as the actual implementation of the switch statement, consists of :

2 memory accesses : one to retrieve the value of the next instruction, one to retrieve the address of the instruction's implementation,

plus 2 branches : one to jump to the bytecode's implementation and another one to go back to the beginning of the loop. Jumps are among the most expensive instructions on modern architectures.

Pure bytecode interpreters are easy to write and to understand. They are also highly portable but rather slow. They are thus not convenient for embedded systems. In the case where most bytecodes perform simple operations, as in the example illustrated herein before, most of the execution time is wasted in performing the dispatch. Actually, in order to be aware of the real cost of the mechanism, it should be compared with the cost of the execution of a single bytecode. Java bytecodes have a very low-level semantics, and their implementation is often trivial. Therefore, the most commonly executed bytecodes are actually less expensive than the dispatching mechanism itself.

A first improvement in efficiency according to the invention is the adoption of indirect threaded code as illustrated with the set of instructions below :

```
Op_1_lbl:
                  // op_1's implementation
          goto opcode_table (*pc++);
Op_2_lbl:
                  // op_2's implementation
          goto opcode_table (*pc++);
Op_3_lbl:
                  // op_3's implementation
          goto opcode_table (*pc++);
```

where Op_1_ lbl, Op_2_ lbl and Op_3_ lbl represent 3 different operation codes interpreted by the VM interpreter.

According to this implementation, called indirect threaded code, the VM is coded as an indirect threaded code interpreter. During bytecode translation, the address of the next bytecode is resolved. A reference table, denoted opcode_table, contains the bytecodes implementation addresses. The reference table is accessed by an index of a pointer (*pc++). For each bytecode translation, the address of the next bytecode is retrieved to jump to the next bytecode. In this way each bytecode implementation directly jumps to the next bytecode implementation, we have saved one branch, the outer loop, and the unnecessary inefficiency of the switch statement's implementation (range checking and default case handling).

According to a preferred embodiment of the invention, the translation is carried out by exploiting unused bytecodes of the bytecode-based language VM specification.

The bloc diagram of figure 1 summarizes the mains steps of the method according to the invention for translating a bytecode, e.g. the bytecode bipush, into native instructions with an indirect threaded code interpreter :

step K0= BIPUSH ; beginning of the method of translating the bytecode bipush which consists of putting ½ word on a stack, the ½ word being the bipush parameter (par)

step K1= PAR ; retrieve the bipush parameter (par)

step K2= PUT; put the bipush parameter on the stack

step K3 = GOTO; go to the next bytecode (goto opcode_table (*pc)) by looking into a reference table opcode_table containing the address of the next bytecode's implementation.

The adoption of threaded code by itself can double the VM's performance, but as we will see in the following it can also offer other interesting optimization opportunities. A statistic analysis of Java's bytecodes shows that, on average, about every 5-6 instructions there is a branch. On any modern CPU, branches are intrinsically expensive instructions, since they can cause pipeline stalls and/or trigger external bus activity. Besides, for loop unrolling or method call in-lining, there is not much that can really be done about it. Even when recompiling the code into a native representation, the control statements will still be there.

Recent studies on the CPU utilization for object oriented applications on high-end workstations show that the CPU can spend as much as 70% of its clock cycles to recover from pipeline stalls, as the effect of mispredicted branch statements, and to wait for data and instructions from a main memory (cache misses). Additionally, CPUs available in embedded systems normally have very small caches, no hardware assistance for dynamic branch prediction, and low and/or narrow memory interfaces with no L2 caches. These additional constraints will reduce even further the CPU utilization and performance.

Java bytecodes can be separated into two categories :

simple operation codes (loads, stores, arithmetic and control statements) and

complex operation codes (memory management, synchronization, etc.).

Simple bytecodes are typically less expensive than the dispatching mechanism. Complex bytecodes are instead much more expensive, the dispatching cost representing only a minimal fraction of the total cost of the bytecode execution cost. Simple bytecodes are also executed much more frequently (about an order of magnitude) than complex ones, implying that a classical Java interpreter spends most of its time dispatching bytecodes rather than really doing anything useful. It is thus assumed that it would be definitively more effective to reduce the dispatching cost for simple bytecodes than for complex ones.

Translating bytecodes into indirect threaded code also gives the opportunity to make arbitrary transformations on the executable code. One such transformation is to detect common sequences of bytecodes and translate them into a single threaded “macro code “. This macro code performs the work of the entire sequence of original bytecodes. Therefore, according to a preferred embodiment of the invention, it is proposed to replace sequences of simple bytecodes by some equivalent "macro codes". For example, as presented in the cited article, the bytecodes “push literal, push variable, add, store variable” can be translated into a single “add-literal-to-variable” macro code in the indirect threaded code. Such optimization are effective because they avoid the overhead of the multiple dispatches that are implied by the original bytecodes, but elided within the macro code. A single macro code which is translated from a sequence of N original bytecodes avoids N-1 bytecode dispatches at execution time. More details about how to build macro codes can be found in the cited article. Such macro codes will have to satisfy the following criteria :

Macros have to be made out of sequences of simple bytecodes, since there is no point in reducing the dispatching cost of complex ones.

Macros must not contain instructions that are possible branch targets, otherwise one would have to radically change the VM execution mechanism. A macro itself can be a branch target.

Macros must terminate with control statements or method calls, since the cost of a native branch is equivalent to that of a dispatch operation.

For implementation simplicity, the maximal length of a macro should be approximately of 15 bytecodes. The "natural" average macro length being of 4-5 bytecodes. From these criteria it is very simple to construct such macro sequences, taking very little -and bounded- CPU time. A simple scan of a method's bytecode is indeed enough, and most of the parsing can be table driven and single-bytecode based.

According to a particular alternative of the preferred embodiment, which takes into account that unused bytecodes are very few (30-40 on average) a two-byte representation can be used for the new bytecodes representing the new macro-instruction. The operands of the original sequence are grouped right after the new sequence, which leaves them easily accessible by incrementing the program counter of the virtual machine.

Once a process is scanned, macros can be constructed by simply cutting and pasting together the binary code produced by the compiler for the threaded code interpreter. Macros are just considered as normal bytecodes by the threading dispatcher.

Figure 2 summarizes the preferred embodiment of a virtual machine according to the invention. The VM is implemented to load programs containing bytes codes to be interpreted by the VM interpreter. The main steps of the method are the following :

step K0= INIT: initialization of the procedure executed by the VM by loading the programs containing the bytecodes ,

step K1= OPCODE : to retrieve the bytecodes to be interpreted,

step K2= MACRO : replacement of sequences of simple bytecodes with macro bytecodes,

step K3= TRANS : interpretation of the macro bytecodes using the indirect threaded interpreter method as described in figure 1,

step K4= RES : get the result, end of the method.

Statistical analysis performed on execution traces of actual Java applications, show that the typical macro length is of 4-5 bytecodes, and that, after the code transformation, macros can be executed up to five times more often than the remaining bytecodes. The remaining bytecodes are those for whom the implementation is just too complex to be worth in-lining and those which are left behind by taking into account the branch target analysis. The total bytecode dispatching cost may thus be reduced by more than a factor of four. If the dispatching cost originally constituted about 50 % of the total execution cost, it can be significantly reduced by using the invention.

The invention brings out some additional advantages. The processor branch instructions can also be reduced by about a factor of five. Since the code to be executed has been linearized, the performance of the processor's pipeline and memory subsystem may be significantly improved. The actual gain depends on the architecture of the processor for the cost of a pipeline stall and on the memory subsystem architecture for the cost of a cache line fill. On "memory challenged" systems, like most embedded applications, these costs are quite high and definitively worth reducing. The residual dispatching cost essentially depends on the control statements present into the Java code. To fully translate the bytecode into binary code like classical dynamic recompilation, branch statements should be introduced in the executable code. This would have more or less the same cost as the remaining dispatches which are left with.

One of the advantages of macros is that they are generic sequences of bytecodes, and that the probability that one of such sequences can be found elsewhere in the context of another process, or even in the same process, is quite high. Test were made for Java bytecodes. It was found that a significant part of the macros can be reused. Therefore, by taking into account the reuse factor, the memory footprint used by macro code implementation may be reduced. A full translation to binary code would consume at least twice as much memory, and would very likely have only a negligible performance advantage. For instance, assuming that it would be possible to further cut the cost of scheduling by another factor of two, the total observable increment in speed would be very small. Most likely, it is not worth trading against the doubling of memory footprint.

Another advantage of macros is that they do not have any impact on the normal bytecode dispatching mechanism. There is no need to add another execution mechanism to those already existing in the VM. There is no need to distinguish between compiled and non-compiled processes and no need to recur to the weirdness and overhead of native code interfaces.

Object-oriented languages like Java are characterized by the presence of very small units of code. Java processes are also very hard to inline, since they are almost always potentially polymorphic. Therefore, even if a fully optimizing compiler would be able to better map the process execution semantics on the underlying processor architecture, the overhead of the preamble and conclusion of binary translated processes would often suppress any advantage.

To improve execution efficiency, a stack catching technique can be used, which keeps the first three locations of the Java stack inside the processor's register file, reducing considerably the number of memory accesses. The technique exploits the fact that the target processor is a stack machine itself. The original bytecode implementations are substituted with equivalent processor instruction sequences. By using a trivial translation table and a simple cost function (number of memory references), very fast and efficient compilation technique can be achieved. The cost reduction of memory Input / output will now be described, in the case of Java as an example, according to another alternative embodiment of the invention.

Java is a stack-based language: bytecodes communicate with each other using memory. Every single bytecode execution implies at least one memory access, which turns out to be very expensive. Considering, for instance, the following simple expression :

C = a + b;

In a stack based language it is translated into :

```
Push  a        -- 1 read,        1 write
Push b         -- 1 read,        1 write
Add            -- 2 read,        1 write
Store c        -- 1 read,        1 write
```

which represents nine memory access operations. A CPU with a minimum of internal state can do the same with only three memory accesses. Considering the fact that on a modern processor architecture, memory references are among the most expensive operations, it is an ideal field of optimization. With a little additional coding effort, a version of the Java bytecodes can be made to exchange data through machines registers instead than through external memory. Macros can then be created, starting from these specialized bytecodes which are called strands, reducing the number of memory accesses within a macro by more than a factor of two.

An implementation of the "macroizer" and of the bytecode "standifier" would not need too many lines of code. Partial rewrite of the interpreter's loop can be estimated, for example, in about a few Kilo lines of C code. Only a few lines of assembly are necessary for the implementation of the indirect threaded code dispatcher, and a few hundreds are dedicated to the "standifier".

Tests and measures of the running time have been made which don't take into account the time spent for the bytecode parsing and for the generation of the new macro bytecodes. Nevertheless the run-time was measured using a native code profiler. When running a large application, like a web browser, the total time spent for "macroization" remains limited to a very little percentage of the total execution time.

An example of a receiver according to the invention is shown in fig. 2. It is a set top box receiver 20 for interactive video transmission. It comprises a decoder, e.g. compatible with the MPEG 2 (Moving Pictures Experts group, ISO/IEC 13818-2) recommendation, for receiving via a cable transmission channel 23 an encoded signal from a video transmitter 24 and for decoding the received signal in order to retrieve the transmitted data to be displayed on a video display 25. The functions of the set top box can be efficiently software implemented using a system that executes an interpreted language such as Java in the form of bytecodes. The system comprises a main processor CPU and a memory MEM for storing software code portions representing instructions for causing the main processor CPU to carry out the methods according to the invention as described in figure 1 or 2.

According to another embodiment of the invention, the set top box 20 can receive Java applications containing bytecodes as part of the received signal. In this case, the set top box would comprise a loader to load the bytecode-based programs received from a distant sender.

CLAIMS:

1. A method of optimizing interpreted programs in a virtual machine interpreter of a bytecode-based language, wherein the virtual machine dynamically reconfigures itself by replacing an original sequence of simple bytecodes with a new sequence of macro bytecodes and wherein the virtual machine interpreter is coded as a threaded code interpreter for translating the bytecodes into their implementation code, comprising a reference table which contains references to the addresses of the implementation of the bytecodes in order that during translation of the current bytecode, the address of the implementation of the next bytecode is retrieved to be able to jump to the next bytecode.

2. A method according to claim 1, wherein the bytecodes of the original sequence are grouped after the new sequence of said macro operation codes.

3. A method according to any of claims 1 or 2, wherein the virtual machine interpreter comprises a predetermined set of bytecodes, some of which are unused, and wherein said new sequence of macro operation codes is implemented by exploiting said unused bytecodes.

4. A method according to claim 3, wherein the unused bytecodes are encoded with at least a two-byte representation.

5. A method of optimizing interpreted programs, in a virtual machine for a bytecode-based language, comprising the following steps :

initialization by loading programs containing the bytecodes,

replacement of sequences of simple bytecodes with macro codes,

interpretation of the macro bytecodes using an indirect threaded interpreter for translating the bytecodes into their implementation code, comprising a reference table which contains references to the addresses of the implementation of the bytecodes in order that during interpretation of the current bytecode, the address of the implementation of the next bytecode is retrieved to be able to jump to the next bytecode.

6. A computer program product for being loaded into a memory, comprising a set of instructions for causing a processor to carry out the method according to any one of claims 1 to 5.

7. A receiver for receiving transmission signals, the receiver comprising a processor (CPU) and a memory (MEM) for storing software code portions representing instructions for causing the processor to carry out the method according to any one of claims 1 to 5.

8. A method of making available for downloading a computer program comprising instructions for executing the method as claimed in any one of the claims 1 to 5, into a receiver as claimed in claim 7.

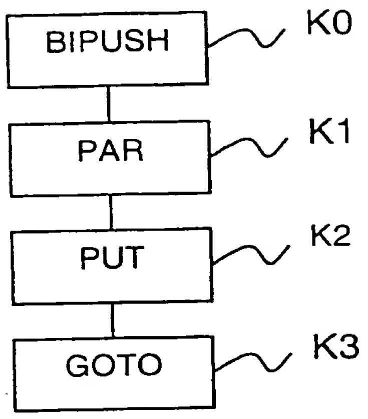


FIG. 1

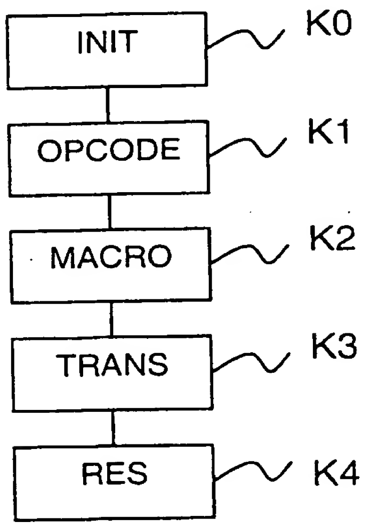


FIG. 2

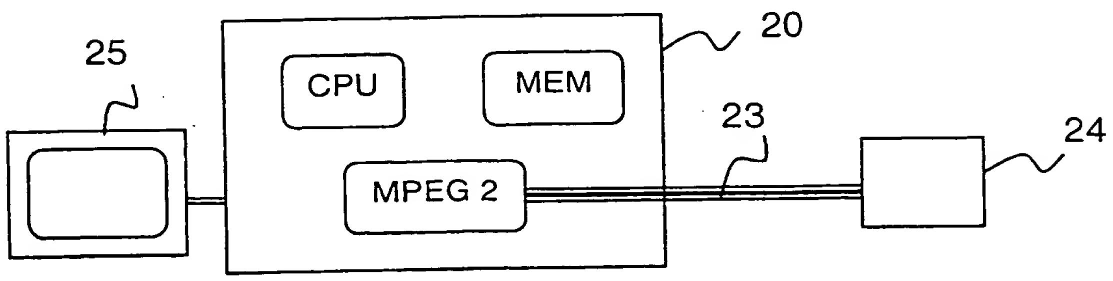


FIG. 3

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 March 2001 (29.03.2001)

PCT

(10) International Publication Number
WO 01/22213 A3

(51) International Patent Classification⁷: G06F 9/45

(21) International Application Number: PCT/EP00/08976

(22) International Filing Date: 13 September 2000 (13.09.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
99402309.1 21 September 1999 (21.09.1999) EP

(71) Applicant: KONINKLIJKE PHILIPS ELECTRONICS N.V. [NL/NL]: Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(72) Inventor: RICCARDI, Fabio: Prof. Holstlaan 6, NL-5581 AW Eindhoven (NL).

(74) Agent: CHARPAIL, François: Internationaal Octrooibureau B.V., Prof. Holstlaan 6, NL-5581 AW Eindhoven (NL).

(81) Designated States *(national)*: CN, IN, JP, KR.

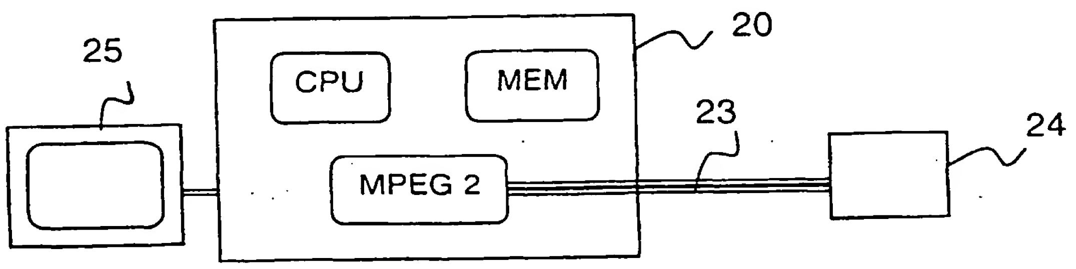
(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *with international search report*

(88) Date of publication of the international search report: 29 November 2001

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: OPTIMIZED BYTECODE INTERPRETER OF VIRTUAL MACHINE INSTRUCTIONS



(57) Abstract: The invention relates to a method of optimizing interpreted programs, in a virtual machine interpreter of a bytecode-based language, comprising means for dynamically reconfiguring said virtual machine with macro operation codes by replacing an original sequence of simple operation codes with a new sequence of said macro operation codes. The virtual machine interpreter is coded as an indirect threading interpreter thanks to a translation table containing the implementation addresses of the operation codes for translating the bytecodes into the operation code's implementation addresses. Application: embedded systems using any bytecode-based programming language, set to box for interactive video transmissions.

WO 01/22213 A3

INTERNATIONAL SEARCH REPORT

ational Application No
PCT/EP 00/08976

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/45

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	PIUMARTA I ET AL: "OPTIMIZING DIRECT THREADED CODE BY SELECTIVE INLINING" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 33, no. 5, 1 May 1998 (1998-05-01), pages 291-300, XP000766278 ISSN: 0362-1340 cited in the application the whole document	1,2,5-8
A	US 5 778 232 A (GROSS DAVID HENRY ET AL) 7 July 1998 (1998-07-07) column 6, line 37 -column 7, line 24	1,5

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search 28 June 2001	Date of mailing of the international search report 06/07/2001
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Bijn, K

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

ational Application No
PCT/EP 00/08976

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5778232 A	07-07-1998	NONE	